The Burrows-Wheeler Transform (1994; M. Burrows, D. J. Wheeler)

Paolo Ferragina, University of Pisa, www.di.unipi.it/~ferragin
Giovanni Manzini, University of Piemonte Orientale, www.mfn.unipmn.it/~manzini

Entry Editor: Paolo Ferragina

INDEX TERMS: Data transform, Lossless data compression, Empirical Entropy, Suffix array.
SYNONYMS: Block-sorting data compression.

1 PROBLEM DEFINITION

The Burrows-Wheeler transform is a technique used for the lossless compression of data. It is the algorithmic core of the tool bzip2 which has become a standard for the creation and distribution of compressed archives.

Before the introduction of the Burrows-Wheeler transform, the field of lossless data compression was dominated by two approaches (see [1, 15] for comprehensive surveys). The first approach dates back to the pioneering works of Shannon and Huffman, and it is based on the idea of using shorter codewords for the more frequent symbols. This idea has originated the techniques of Huffman and Arithmetic Coding, and, more recently, the PPM (Prediction by Partial Matching) family of compression algorithms. The second approach originated from the works of Lempel and Ziv and is based on the idea of adaptively building a dictionary and representing the input string as a concatenation of dictionary words. The best known compressors based on this approach form the so called ZIP-family; they have been the standard for several years and are available on essentially any computing platform (e.g. gzip, zip, winzip, just to cite a few).

The Burrows-Wheeler transform introduced a completely new approach to lossless data compression based on the idea of *transforming* the input to make it easier to compress. In the authors' words: "(this) technique [...] works by applying a reversible transformation to a block of text to make redundancy in the input more accessible to simple coding schemes" [3, Sect. 7]. Not only this technique produced some state-of-the-art compressors, but it also originated the field of compressed indexes [14] and it has been successfully extended to compress (and index) structured data such as XML files [7] and tables [16].

2 KEY RESULTS

Notation. Let s be a string of length n drawn from an alphabet Σ. For $i = 0, \ldots, n-1$, $s[i]$ denotes the i-th character of s, and $s[i, n-1]$ denotes the suffix of s starting at position i (that is, starting with the character $s[i]$). Given two strings s and t, the notation $s \prec t$ is used to denote that s lexicographically precedes t.

The Burrows-Wheeler Transform. In [3] Burrows and Wheeler introduced a new compression algorithm based on a reversible transformation, now called the *Burrows-Wheeler Transform* (shortly, bwt). Given a string s, the computation of bwt(s) consists of three basic steps (see Fig. 1):

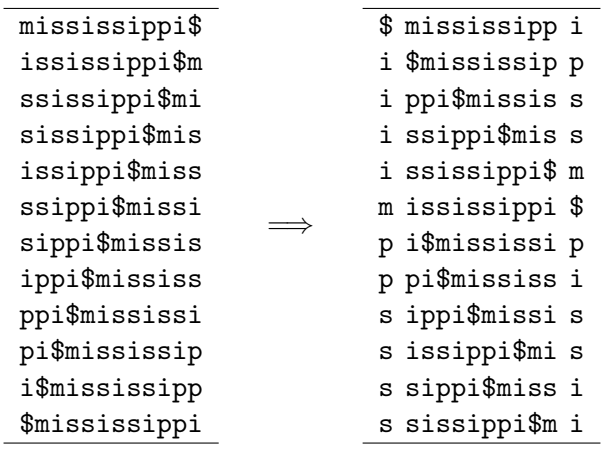

Figure 1: Example of Burrows-Wheeler transform for the string $s = \texttt{mississippi}$. The matrix on the right has the rows sorted in lexicographic order. The output of the bwt is the last column of the sorted matrix; in this example the output is $\hat{s} = \mathsf{bwt}(s) = \texttt{ipssm\$pissii}$.

1. append to the end of s a special symbol \$ smaller than any other symbol in Σ;
2. form a *conceptual* matrix $\mathcal{M}$ whose rows are the cyclic shifts of the string $s\$$ sorted in lexicographic order;
3. construct the transformed text $\hat{s} = \mathsf{bwt}(s)$ by taking the last column of $\mathcal{M}$.

Notice that every column of $\mathcal{M}$, hence also the transformed text $\hat{s}$, is a permutation of $s\$$. As an example F, the first column of the bwt matrix $\mathcal{M}$, consists of all characters of s alphabetically sorted. In Fig. 1 it is $\mathsf{F} = \texttt{\$iiiimppssss}$.

Although it is not obvious from its definition, the bwt is an invertible transformation and both the bwt and its inverse can be computed in $O(n)$ optimal time. To be consistent with the more recent literature, the following notation and proof techniques will be slightly different from the ones in [3].

Definition 1. *For $1 \leq i \leq n$, let $s[k_i, n-1]$ denote the suffix of s prefixing row i of $\mathcal{M}$, and define $\Psi(i)$ as the index of the row prefixed by $s[k_i + 1, n-1]$.*

For example in Fig. 1 it is $\Psi(2) = 7$ since row 2 of $\mathcal{M}$ is prefixed by `ippi` and row 7 is prefixed by `ppi`. Note that $\Psi(i)$ is not defined for $i = 0$ since row 0 is not prefixed by a proper suffix of s.[1]

Lemma 1. *For $i = 1, \ldots, n$, it is $\mathsf{F}[i] = \hat{s}[\Psi(i)]$.*

Proof. Since each row contains a cyclic shift of $s\$$, the last character of the row prefixed by $s[k_i + 1, n-1]$ is $s[k_i]$. Definition 1 then implies $\hat{s}[\Psi(i)] = s[k_i] = \mathsf{F}[i]$ as claimed. □

Lemma 2. *If $1 \leq i < j \leq n$ and $\mathsf{F}[i] = \mathsf{F}[j]$ then $\Psi(i) < \Psi(j)$.*

Proof. Let $s[k_i, n-1]$ (resp. $s[k_j, n-1]$) denote the suffix of s prefixing row i (resp. row j). The hypothesis $i < j$ implies that $s[k_i, n-1] \prec s[k_j, n-1]$. The hypothesis $\mathsf{F}[i] = \mathsf{F}[j]$ implies $s[k_i] = s[k_j]$ hence it must be $s[k_i + 1, n-1] \prec s[k_j + 1, n-1]$. The thesis follows since by construction $\Psi(i)$ (resp. $\Psi(j)$) is the lexicographic position of the row prefixed by $s[k_i + 1, n-1]$ (resp. $s[k_j + 1, n-1]$). □

[1] In [3] instead of Ψ the authors make use of a map which is essentially the inverse of Ψ. The use of Ψ has been introduced in the literature of compressed indexes where Ψ and its inverse play an important role (see [14]).

Lemma 3. *For any character $c \in \Sigma$, if $\mathsf{F}[j]$ is the ℓ-th occurrence of c in F, then $\hat{s}[\Psi(j)]$ is the ℓ-th occurrence of c in $\hat{s}$.*

Proof. Take an index h such that $h < j$ and $\mathsf{F}[h] = \mathsf{F}[j] = c$ (the case $h > j$ is symmetric). Lemma 2 implies $\Psi(h) < \Psi(j)$ and Lemma 1 implies $\hat{s}[\Psi(h)] = \hat{s}[\Psi(j)] = c$. Consequently, the number of c's preceding (resp. following) $\mathsf{F}[j]$ in F coincides with the number of c's preceding (resp. following) $\hat{s}[\Psi(j)]$ in $\hat{s}$ and the lemma follows. □

In Fig. 1 it is $\Psi(2) = 7$ and both $\mathsf{F}[2]$ and $\hat{s}[7]$ are the second `i` in their respective strings. This property is usually expressed by saying that corresponding characters maintain the *same relative order* in both strings F and $\hat{s}$.

Lemma 4. *For any i, $\Psi(i)$ can be computed from $\hat{s} = \mathsf{bwt}(s)$.*

Proof. Retrieve F simply by sorting alphabetically the symbols of $\hat{s}$. Then compute $\Psi(i)$ as follows: (1) set $c = \mathsf{F}[i]$, (2) compute ℓ such that $\mathsf{F}[i]$ is the ℓ-th occurrence of c in F, (3) return the index of the ℓ-th occurrence of c in $\hat{s}$. □

Referring again to Fig. 1, to compute $\Psi(10)$ it suffices to set $c = \mathsf{F}[10] = \mathtt{s}$ and observe that $\mathsf{F}[10]$ is the second `s` in F. Then it suffices to locate the index j of the second `s` in $\hat{s}$, namely $j = 4$. Hence $\Psi(10) = 4$, and in fact row 10 is prefixed by `sissippi` and row 4 is prefixed by `issippi`.

Theorem 5. *The original string s can be recovered from $\mathsf{bwt}(s)$.*

Proof. Lemma 4 implies that the column F and the map Ψ can be retrieved from $\mathsf{bwt}(s)$. Let j_0 denote the index of the special character \$ in $\hat{s}$. By construction, the row j_0 of the bwt matrix is prefixed by $s[0, n-1]$, hence $s[0] = \mathsf{F}[j_0]$. Let $j_1 = \Psi(j_0)$. By Definition 1 row j_1 is prefixed by $s[1, n-1]$ hence $s[1] = \mathsf{F}[j_1]$. Continuing in this way it is straightforward to prove by induction that $s[i] = \mathsf{F}[\Psi^i(j_0)]$, for $i = 1, \ldots, n-1$. □

Algorithmic issues. A remarkable property of the bwt is that both the direct and the inverse transform admit efficient algorithms that are extremely simple and elegant.

Theorem 6. *Let $s[1, n]$ be a string over a constant size alphabet Σ. String $\hat{s} = \mathsf{bwt}(s)$ can be computed in $O(n)$ time using $O(n \log n)$ bits of working space.*

Proof. The Suffix Array of s can be computed in $O(n)$ time and $O(n \log n)$ bits of working space by using for example the algorithm in [11]. The Suffix Array is an array of integers $\mathsf{sa}[1, n]$ such that for $i = 1, \ldots, n$, $s[\mathsf{sa}[i], n-1]$ is the i-th suffix of s in the lexicographic order. Since each row of $\mathcal{M}$ is prefixed by a unique suffix of s followed by the special symbol \$, the Suffix Arrays provides the ordering of the rows in $\mathcal{M}$. Consequently, $\mathsf{bwt}(s)$ can be computed from sa in linear time using the procedure $\mathsf{sa2bwt}$ of Fig. 2. □

Theorem 7. *Let $s[1, n]$ be a string over a constant size alphabet Σ. Given $\mathsf{bwt}(s)$, the string s can be retrieved in $O(n)$ time using $O(n \log n)$ bits of working space.*

Proof. The algorithm for retrieving s follows almost verbatim the procedure outlined in the proof of Theorem 5. The only difference is that, for efficiency reasons, all the values of the map Ψ are computed in one shot. This is done by the procedure $\mathsf{bwt2psi}$ in Fig. 2. In $\mathsf{bwt2psi}$ instead of working with the column F, it is used the array count which is a "compact" representation of F. At the beginning of the procedure, for any character $c \in \Sigma$, $\mathsf{count}[c]$ provides the index of the first row of $\mathcal{M}$ prefixed by c. For example, in Fig. 1 $\mathsf{count}[\mathtt{i}] = 1$, $\mathsf{count}[\mathtt{m}] = 5$, and so on. In the main `for` loop of $\mathsf{bwt2psi}$ the array bwt is scanned and $\mathsf{count}[c]$ is increased every time an occurrence of character c is encountered (line 6). Line 6 also assigns to `h` the index of the ℓ-th occurrence of c in F.

Procedure sa2bwt

```
1. bwt[0]=s[n-1];
2. for(i=1;i<=n;i++) {
3.   if(sa[i] == 1)
4.     bwt[i]='$';
5.   else
6.     bwt[i]=s[sa[i]-1];
7. }
```

Procedure bwt2psi

```
1. for(i=0;i<=n;i++) {
2.   c = bwt[i];
3.   if(c == '$')
4.     j0 = i;
5.   else {
6.     h = count[c]++;
7.     psi[h]=i;
8.   }
9. }
```

Procedure psi2text

```
1. k = j0; i=0;
2. do {
3.   k = psi[k];
4.   s[i++] = bwt[k];
5. } while(i<n);
```

Figure 2: Algorithms for computing and inverting the Burrows-Wheeler Transform. Procedure sa2bwt computes bwt(s) given s and its suffix array sa. Procedure bwt2psi takes bwt(s) as input and computes the Ψ map storing it in the array `psi`. bwt2psi also stores in `j0` the index of the row prefixed by $s[0, n-1]$. bwt2psi uses the auxiliary array `count`$[1, |\Sigma|]$ which initially contains in `count[i]` the number of occurrences in bwt(s) of the symbols $1, \ldots, i-1$. Finally, procedure psi2text recovers the string s given bwt(s), the array `psi`, and the value j_0.

By Lemma 3, line 7 stores correctly in psi$[h]$ the value $i = \Psi(h)$. After the computation of array psi, s is retrieved by using the procedure psi2text of Fig. 2, whose correctness follows immediately by Theorem 5.

Clearly the procedures bwt2psi and psi2text in Fig. 2 run in $O(n)$ time. Their working space is dominated by the cost of storing the array psi which takes $O(n \log n)$ bits. □

The Burrows-Wheeler compression algorithm. The rationale for using the bwt for data compression is the following. Consider a string w that appears k times within s. In the bwt matrix of s there will be k consecutive rows prefixed by w, say rows $r_w + 1, r_w + 2, \ldots, r_w + k$. Hence, the positions $r_w + 1, \ldots, r_w + k$ of $\hat{s} =$ bwt(s) will contain precisely the symbols that immediately precede w in s. If in s certain patterns are more frequent than others, then for many substrings w the corresponding positions $r_w + 1, \ldots, r_w + k$ of $\hat{s}$ will contain only a few distinct symbols. For example, if s is an English text and w is the string `his`, the corresponding portion of $\hat{s}$ will likely contain many `t`'s and blanks and only a few other symbols. Hence $\hat{s}$ is a permutation of s that is usually *locally homogeneous*, in that its "short" substrings usually contain only few distinct symbols.[2]

To take advantage of this property, Burrows and Wheeler proposed to process the string $\hat{s}$ using move-to-front encoding [2] (shortly, mtf). mtf encodes each symbol with the number of distinct symbols encountered since its previous occurrence. To this end, mtf maintains a list of the symbols ordered by recency of occurrence; when the next symbol arrives the encoder outputs its current rank and moves it to the front of the list. Note that mtf produces a string which has the same length as $\hat{s}$ and, if $\hat{s}$ is locally homogeneous, the string mtf($\hat{s}$) will mainly consist of small integers.[3] Given this skewed distribution, mtf($\hat{s}$) can be easily compressed: Burrows and Wheeler proposed to compress it using Huffman or Arithmetic coding, possibly preceded by the run-length encoding of runs of equal integers.

Burrows and Wheeler were mainly interested in proposing an algorithm with good practical performance. Indeed their simple implementation outperformed, in terms of compression ratio, the tool gzip that was the current standard for lossless compression. A few years after the introduction of the bwt, [9, 12] have shown that the compression ratio of the Burrows-Wheeler compression algorithm can be bounded in terms of the k-th order empirical entropy of the input string for any

[2] Obviously this is true only if s has some regularity: if s is a random string $\hat{s}$ will be random as well!

[3] If s is an English text, mtf($\hat{s}$) usually contains more that 50% zeroes.

$k \geq 0$. For example, Kaplan et al. [9] showed that for any input string s and real $\mu > 1$, the length of the compressed string is bounded by $\mu n H_k(s) + n \log(\zeta(\mu)) + \mu g_k + O(\log n)$ bits, where $\zeta(\mu)$ is the standard Zeta function and g_k is a function depending only on k and Σ. This bound holds *pointwise* for *any* string s, *simultaneously* for any $k \geq 0$ and $\mu > 1$, and it is remarkable since similar bounds have not been proven for any other known compressor. The theoretical study on the performance of bwt-based compressors is currently a very active area of research. The reader is referred to the recommended readings for further information.

3 APPLICATIONS

After the seminal paper of Burrows and Wheeler, many researchers have proposed compression algorithms based on the bwt (see [4, 5] and references therein). Of particular theoretical interest are the results in [6] showing that the bwt can be used to design a "compression booster", that is, a tool for improving the performance of other compressors in a well defined and measurable way.

Another important area of application of the bwt is the design of Compressed Full-text Indexes [14]. These indexes take advantage of the relationship between the bwt and the Suffix Array to provide a compressed representation of a string supporting the efficient search and retrieval of the occurrences of an arbitrary pattern.

4 OPEN PROBLEMS

In addition to the investigation on the performance of bwt-based compressors, an open problem of great practical significance is the space efficient computation of the bwt. Given a string s of length n over an alphabet Σ, both s and $\hat{s} = \mathsf{bwt}(s)$ take $O(n \log |\Sigma|)$ bits. Unfortunately, the linear time algorithms shown in Fig. 2 make use of auxiliary arrays (i.e. sa and psi) whose storage takes $\Theta(n \log n)$ bits. This poses a serious limitation to the size of the largest bwt that can be computed in main memory. Space efficient algorithms for inverting the bwt have been obtained in the compressed indexing literature [14], while the problem of space and time efficient computation of the bwt is still open even if interesting preliminary results are reported in [8, 10, 13].

5 EXPERIMENTAL RESULTS

An experimental study of the performance of several compression algorithms based on the bwt and a comparison with other state-of-the-art compressors is presented in [4].

6 DATA SETS

The data set used in [4] are available from http://www.mfn.unipmn.it/~manzini/boosting. Other data sets relevant for compression and compressed indexing are available at the Pizza&Chili site http://pizzachili.di.unipi.it/.

7 URL to CODE

The Compression Boosting page (http://www.mfn.unipmn.it/~manzini/boosting) contains the source code of the algorithms tested in [4]. A more "lightweight" code for the computation of the bwt and its inverse (without compression) is available at http://www.mfn.unipmn.it/~manzini/lightweight. The code of bzip2 is available at http://www.bzip.org.

8 CROSS REFERENCES

Arithmetic coding, Boosting Textual Compression, Compressed text indexing, Compressed suffix array, Suffix array construction, Table compression, Tree compression and indexing.

9 RECOMMENDED READING

[1] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text compression*, Prentice Hall, NJ, 1990.

[2] J. Bentley, D. Sleator, R. Tarjan, and V. Wei, *A locally adaptive compression scheme*, Communications of the ACM, 29 (1986), pp. 320–330.

[3] M. Burrows and D. Wheeler, *A block sorting lossless data compression algorithm*, Tech. Report 124, Digital Equipment Corporation, 1994.

[4] P. Ferragina, R. Giancarlo, and G. Manzini, *The engineering of a compression boosting library: Theory vs practice in bwt compression*, in Proc. 14th European Symposium on Algorithms (ESA), Springer Verlag LNCS n. 4168, 2006, pp. 756–767.

[5] ——, *The myriad virtues of wavelet trees*, in Proc. 33th International Colloquium on Automata and Languages (ICALP), Springer Verlag LNCS n. 4051, 2006, pp. 561–572.

[6] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino, *Boosting textual compression in optimal linear time*, Journal of the ACM, 52 (2005), pp. 688–713.

[7] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan, *Structuring labeled trees for optimal succinctness, and beyond*, in Proc. 46th IEEE Symposium on Foundations of Computer Science (FOCS), 2005, pp. 184–193.

[8] W. Hon, K. Sadakane, and W. Sung, *Breaking a time-and-space barrier in constructing full-text indices*, in Proc. of the 44th IEEE Symposium on Foundations of Computer Science (FOCS), 2003, pp. 251–260.

[9] H. Kaplan, S. Landau, and E. Verbin, *A simpler analysis of Burrows-Wheeler based compression*, in Proc. of the 17th Symposium on Combinatorial Pattern Matching (CPM), LNCS 4009, Springer-Verlag, 2006.

[10] J. Kärkkäinen, *Fast BWT in small space by blockwise suffix sorting*. Theoretical Computer Science, 2007 (to appear). See also http://dimacs.rutgers.edu/Workshops/BWT/index.html.

[11] J. Kärkkäinen, P. Sanders, and S. Burkhardt, *Linear work suffix array construction*, Journal of the ACM, 53(6), 2006, pp. 918-936.

[12] G. Manzini, *An analysis of the Burrows-Wheeler transform*, Journal of the ACM, 48 (2001), pp. 407–430.

[13] J. Na, *Linear-time construction of compressed suffix arrays using $o(n \log n)$-bit working space for large alphabets*, in Proc. 16th Symposium on Combinatorial Pattern Matching (CPM), Springer-Verlag LNCS n. 3537, 2005, pp. 57–67.

[14] G. Navarro and V. Mäkinen, *Compressed full text indexes*, ACM Computing Surveys, 39(1), 2007.

[15] D. Salomon, *Data Compression: the Complete Reference, 3rd Edition*, Springer Verlag, 2004.

[16] B. D. Vo and K.-P. Vo, *Using column dependency to compress tables*, in Proc. of IEEE Data Compression Conference (DCC), 2004, pp. 92–101.